ECE 385

Spring 2021

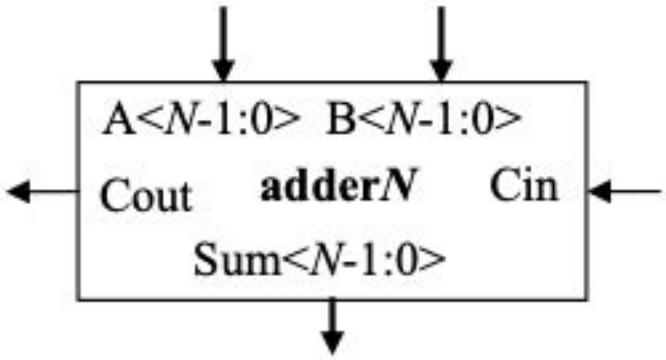
Experiment 3

16-bit Adders

Wenhao Tan & Jiacheng Huang
ABI
Yanye Li

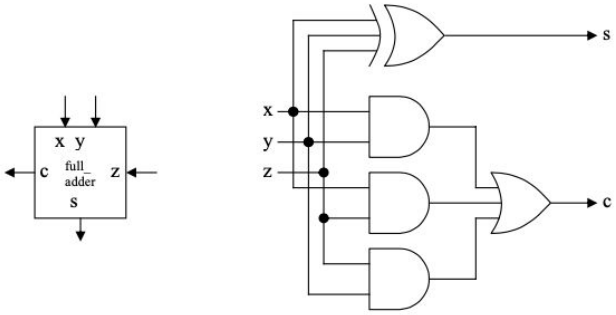
Introduction

In this lab, we transitioned from the CMOS physical logic elements to RTL design on FPGA using SystemVerilog. Specifically, we implemented a carry-ripple adder, a carry-lookahead adder, and a carry-select adder on Quartus. What we implemented can all be abstracted as the block diagram shown below, a 16-bit adder with inputs A[15:0], B[15:0], and Cin, and outputs Sum[15:0] and Cout. All of the three kinds of adders are able to sum up two 16-bit integers, but with different trade-offs on their efficiency, expense, flexibility, and so on.

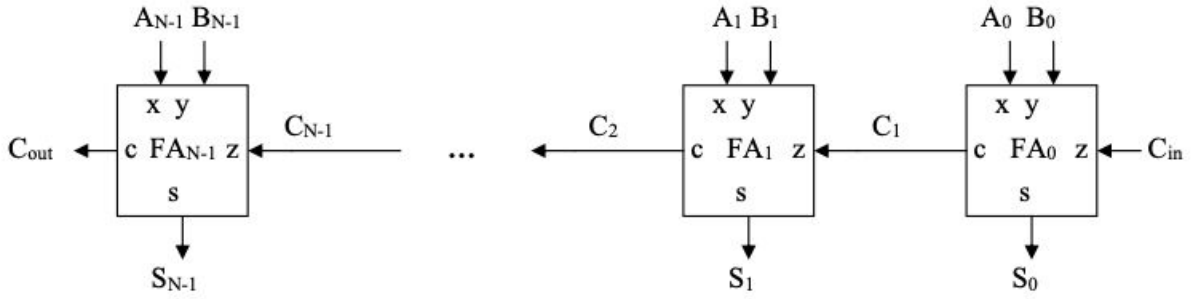


Carry-Ripple Adder

The first adder we implemented was the easiest and most straightforward one - the carry-ripple adder, which is constructed using 16 full-adders, a single-bit binary adder (shown below) that has inputs x, y, and z and outputs s and c.

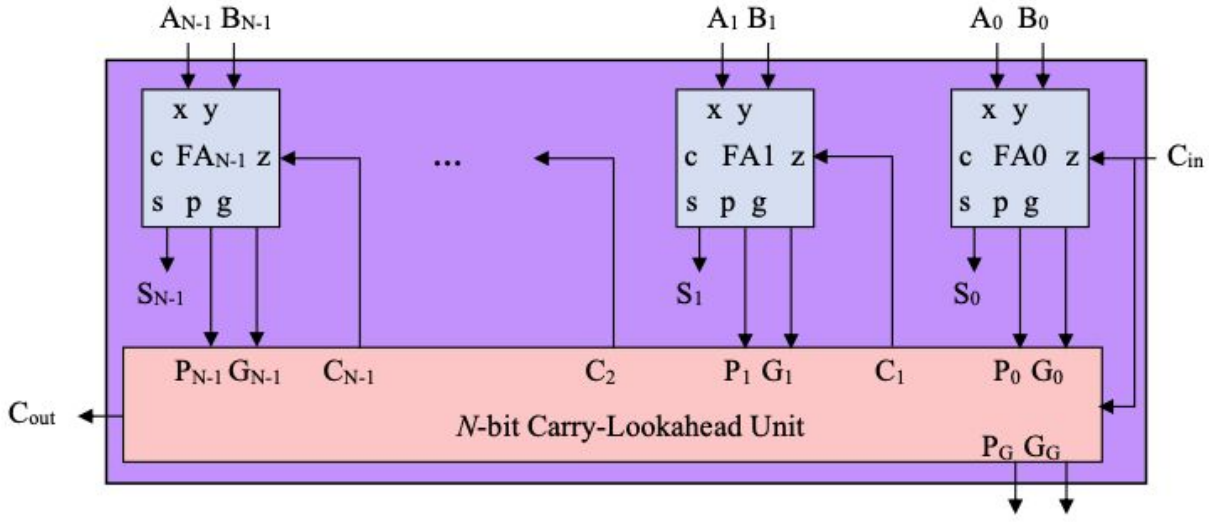


A carry-ripple adder is straightforward, because all we need to do is to simply link them together (as shown below). However, in order to reduce the chance of making mistakes and for the ease of debugging, we divided a 16-bit carry-ripple adder into four 4-bit carry-ripple adder, but they behave exactly the same as being connected directly together.



Carry-Lookahead Adder

From my point of view, the carry-lookahead adder is the most complex one among the three, because it is implemented using a completely new concept – generating(G) and propagating(P). This concept allows us to make predictions on what the carry-outs would be using only inputs A and B, which means we do not have to wait for carry-ins for each bit like what we do in a carry-ripple adder. In this logic, a carry-out is generated if and only if both inputs A and B are 1, and a carry-out is propagated if either A or B is 1. Therefore, a carry-out bit can be expressed as $C_{i+1} = C_i + (P_i * C_i)$. Now we may see a recursive relationship among all carry-ins, if we recursively expand this expression, then for any carry-in bit, there are only Cin, G, and P inside the expressions. Given G = A * B, and P = A XOR B, we are able to find out each carry-in directly from Cin, A and B. However, because the recursion becomes enormously complicated when the number of bits becomes larger, it is convenient to combine four 4-bit carry-lookahead adders into one hierarchical 4x4-bit CLA. For each of the 4-bit CLA, we have the following structure:



Here N = 4, and the internal logic that we used are as follows:

$$C_0 = C_{in}$$
$$C_1 = C_{in} \cdot P_0 + G_0$$
$$C_2 = C_{in} \cdot P_0 \cdot P_1 + G_0 \cdot P_1 + G_1$$
$$C_3 = C_{in} \cdot P_0 \cdot P_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + G_1 \cdot P_2 + G_2$$

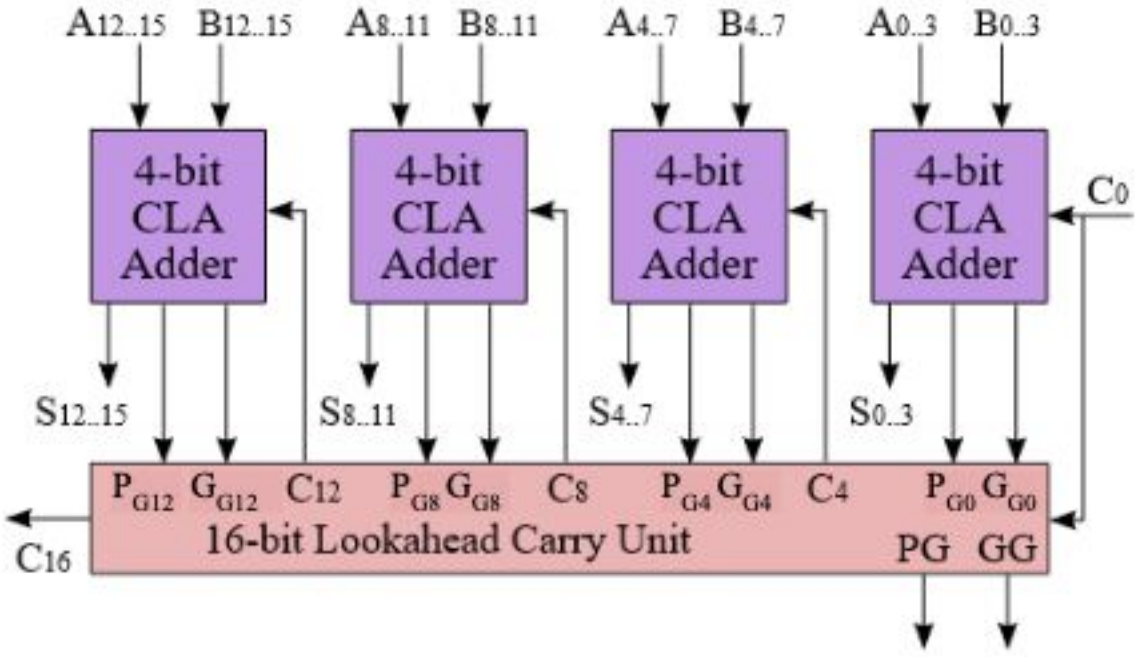
Now for each 4-bit adder, there is a group propagate and group generate logic as shown below:

$$P_G = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$G_G = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

Then we may treat the CLAs as what we did to the full adders, we find the carry-ins for each group using only the group propagate and group generate:

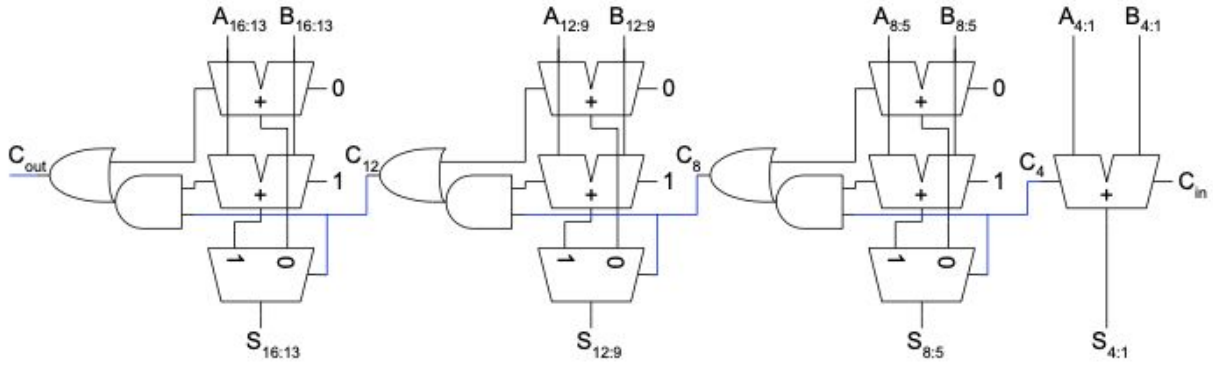
$$C_4 = G_{G0} + C_0 \cdot P_{G0}$$
$$C_8 = G_{G4} + G_{G0} \cdot P_{G4} + C_0 \cdot P_{G0} \cdot P_{G4}$$
$$C_{12} = G_{G8} + G_{G4} \cdot P_{G8} + G_{G0} \cdot P_{G8} \cdot P_{G4} + C_0 \cdot P_{G8} \cdot P_{G4} \cdot P_{G0}$$

Therefore, we connect the four 4-bit CLAs into a 16-bit one without loss of its efficiency, because the connections between them are also implemented using the P and G concept. Here is the diagram for the 4x4-bit hierarchical carry-lookahead adder design:



Carry-Select Adder

A carry-select adder is not as complex as a carry-lookahead adder, because it makes use of full adders as mentioned in the CRA section. However, instead of simply linking the full adders, for each unknown carry-ins, CSA uses two parallel full adders to pre-compute the possible Sums and Couts. Therefore, once the carry-in arrives, the corresponding Sums and Couts can be selected. This process is faster than the CRA, because the propagation speed of the carry-ins only depends on the delays of a small part of logic elements, instead of the delay of a complete full adder. Similarly, we implement the CSA in a 4x4 hierarchical way, as shown below:



Descriptions of .SV Modules

Module: `ripple_adder.sv`

Inputs: `[15:0] A; [15:0] B; cin`

Outputs: `[15:0] S; cout`

Description: This is a 16-bit Carry-Ripple Adder(CRA) created by combining four 4-bit carry-ripple adders, which is combined with four 1-bit full-adders.

Purpose: This module is used to create a 16-bit CRA that calculates the sum of two 16-bit inputs `A` and `B` and stores the output in `S` and the carry out in `cout`.

Module: `lookahead_adder.sv`

Inputs: `[15:0] A; [15:0] B; cin`

Outputs: `[15:0] S; cout`

Description: This is a 16-bit Carry-LookAhead Adder(CLA) created by combining four 4-bit lookahead-adders, which is combined with four 1-bit lookahead-adders. The carry bits in this adder can be computed directly from logic combinations of `cin`, `A`, and `B`.

Purpose: This module is used to create a 16-bit CLA that calculates the sum of two 16-bit inputs `A` and `B` and stores the output in `S` and the carry out in `cout`. The computing time of CLA is much faster than CRA but the space required is more than CRA because it requires many logic gates to store `P`, `G`, `PP`, `GG`, the components to calculate carry out bits.

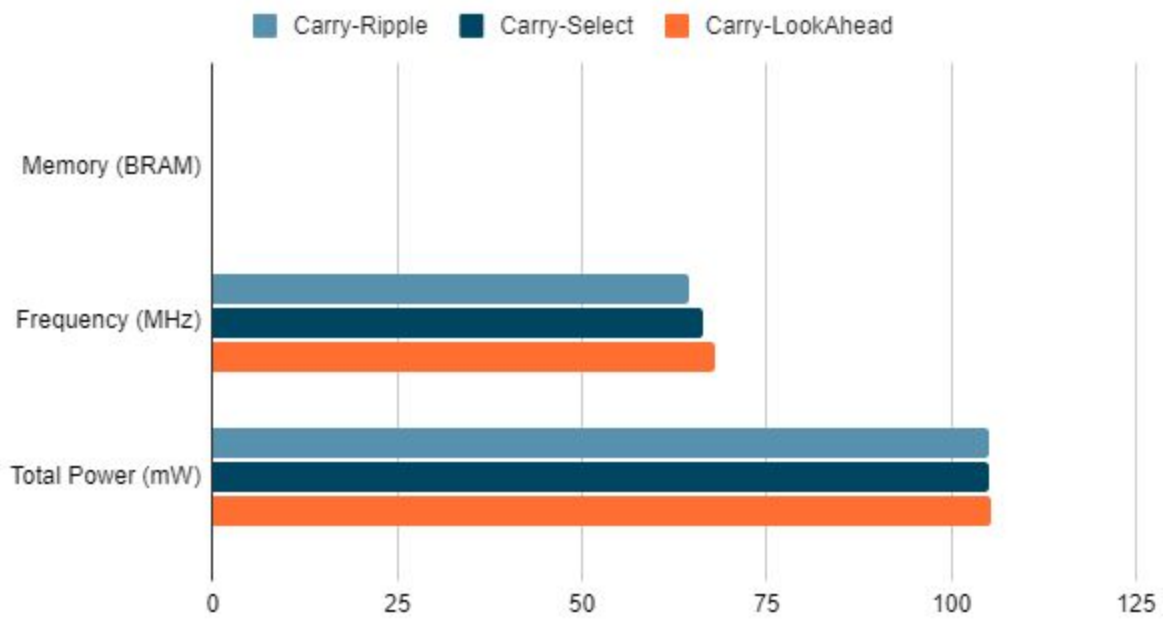
Module: `select_adder.sv`
Inputs: `[15:0] A; [15:0] B; cin`
Outputs: `[15:0] S; cout`
Description: This is a 16-bit Carry-Select Adder(CSA) created by combining seven 4-bit carry-ripple adders with three 2-to-1 MUXs. Both possible outcomes for carry-in 0 and 1 are previously computed, which saves computing time when the carry-in arrives, the MUX can just select the correct adder to proceed.
Purpose: This module is used to create a 16-bit CSA that calculates the sum of two 16-bit inputs `A` and `B` and stores the output in `S` and the carry out in `cout`. The computing time of CSA is much faster than CRA but the space required is more than CRA because it requires almost twice as many components.

Tradeoffs Comparison

We have discussed some tradeoffs in the previous sections, but now let's elaborate. A carry-ripple adder takes the least area (cheapest), has the lowest complexity, and the poorest performance, as it simply connects full-adders together, and needs to wait for carry-ins from the previous full-adder for each bit. A carry-lookahead adder is the most complex and efficient one among them that takes a middle-level area. The performance of a CLA is so fast, because it is able to compute all carry-ins directly from inputs Cin, A, and B, using the concept of propagation and generation, which means each bit of the result is independent of each other, and does not wait for anything. A carry-select adder is faster than a carry-ripple adder, but slower than a carry-lookahead adder, which takes the most area with middle-level complexity. It speeds up by pre-computes the potential results in parallel, so that once the carry-ins arrive, the results can be chosen rapidly, because the dependency is not on complete full adders now, but only on the selection logic. However, because it needs two full-adders except the very first bit, the area that it takes is almost twice that of a carry-ripple adder. The performance differences are shown in the next section.

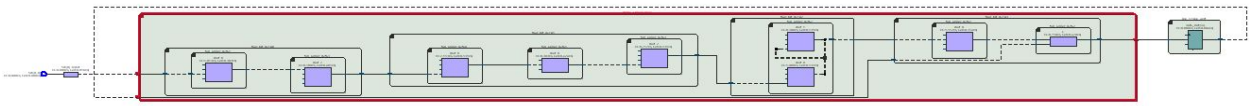
Performance Comparison Plot

	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory (BRAM)	0	0	0
Frequency	64.47 MHz	66.44 MHz	68.0 MHz
Total Power	105.11 mW	105.23 mW	105.36 mW



Critical Path Analysis

Carry-Ripple:

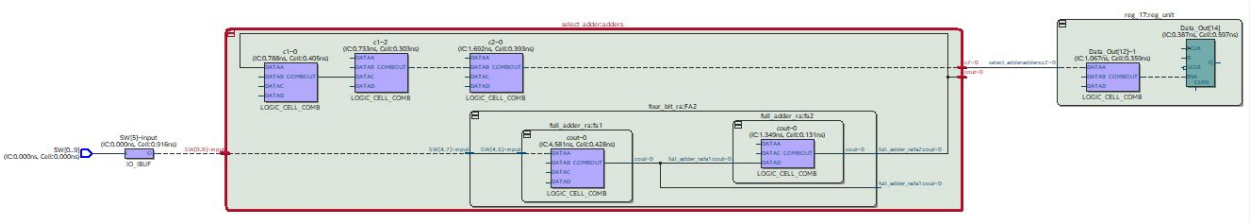


Path #1: Setup slack is 7.204

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	SW[0]
2	To Node	reg_17:reg_unit|Data_Out[14]
3	Launch Clock	Clk
4	Latch Clock	Clk
5	Data Arrival Time	16.178
6	Data Required Time	23.382
7	Slack	7.204

Carry-Select:

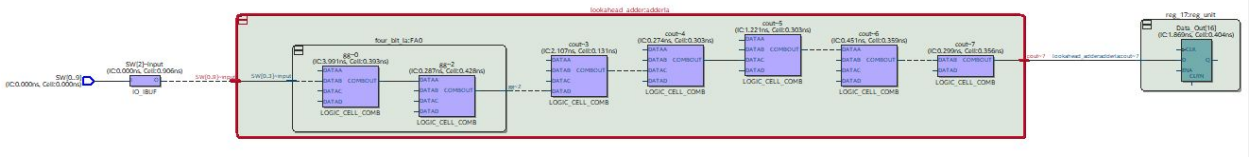


Path #1: Setup slack is 9.264

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	SW[5]
2	To Node	reg_17:reg_unit|Data_Out[14]
3	Launch Clock	Clk
4	Latch Clock	Clk
5	Data Arrival Time	14.129
6	Data Required Time	23.393
7	Slack	9.264

Carry-LookAhead:



Path #1: Setup slack is 8.721

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	SW[2]
2	To Node	reg_17:reg_unit|Data_Out[16]
3	Launch Clock	Clk
4	Latch Clock	Clk
5	Data Arrival Time	14.082
6	Data Required Time	22.803
7	Slack	8.721

Shown above are the critical path analysis diagrams. As we may see, in the CRA, the carry-ins are holding back the design, because the latter full adders are waiting for the previous ones to produce the results before they actually calculate the correct result. The data arrival time for CRA is 16.178.

In the CSA, we are still waiting for the carry-ins in the 4-bit CRAs. However, because the results are pre-computed, once the carry-ins arrive, the results are ready, so the only limiting factor is the selection part of the circuit, that's what makes the circuit different from the first one. The data arrival time is 14.129, which is much faster than CRA.

For the CLA, there is no need to wait for anything, because the carry-ins are calculated simply by logic combinations of Cin, A, and B. Therefore, the CLA is limited only by the logic components, but because this delay is independent and parallel for each bit, the CLA should be

the fastest among the three. The data arrival time is 14.082, which is the fastest. These results are consistent with what we get in other sections of the report.

Answers to The Post-Lab Questions

1) It is not ideal to create a 4x4 hierarchical CSA as in this lab, because we are still using 4-bit CRAs, which slow down the CSA. In order to make it more ideal, we may choose to develop 4-bit carry-select adders first, and then implement the 4x4 hierarchy using the new 4-bit CSAs. In this way, the slow-down caused by the 4-bit CRAs are eliminated.

2) The resulting data plot matches our prediction. The maximum operating frequency of CRA is lower than CLA and CSA because it takes the most time to calculate each carry-out bit one at a time. CLA is fastest because it calculates all carry-out bits previously. CSA is slower because we are implementing it using four 4-bit CRA which are slow. The total power consumption also makes sense. CLA has greater power consumption than CSA, which consumes more than CRA. This is because CLA uses the most logic elements, and CSA uses second most, and CRA uses least.

	Carry-Ripple	Carry-Select	Carry-Lookahead
LUT	78	82	86
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	20	20	20
Frequency	64.47 MHz	66.44 MHz	68.0 MHz
Static Power	89.97 mW	89.97 mW	89.97 mW
Dynamic Power	1.39 mW	1.48 mW	1.57 mW
Total Power	105.11 mW	105.23 mW	105.36 mW

Conclusion

During this lab, we understood the importance of having a consistent naming convention. When we were debugging, sometimes the compiler is not going to tell that a logic wire does not exist. For example, when the port should be Cin, and what I input was cin, there was no compile error, but still there is a run-time error. After finding and replacing all Cin to small cases cin among all files, the problems are solved. Typos are evil, especially when the compiler does not check for

some of them. Therefore, obeying a consistent naming convention is important, which is good for memorizing, understanding, and debugging.

The structure of the lab manual is good, which separates our tasks into logical parts, and explains each adder in detail, so I think it should be unchanged in the future. However, each time we want to test a different type of adder on our FPGA board, we have to recompile the code, which takes a whole minute to complete. I know that the compilation time is not under our control, but we may design a new way to switch between adders using switches, in which way we don't have to recompile for testing all the adders.

In a word, we have learnt a lot about different types of adders and their trade-offs, and the workflow to develop hardware on the FPGA board in SystemVerilog using Quartus.